Variant Selection in Adaptive Parallel Algorithms

John E. West
CEWES MSRC
USAE Waterways Experiment Station
3909 Halls Ferry Road
Vicksburg, MS 39180

Abstract

Scientific and engineering applications in the DOD require high performance to meet operational and research objectives, while the dynamic and diverse HPC environments in the Major Shared Resource Centers require that applications also be portable. The challenge in programming for either performance or portability is significant, and these difficulties increase as demands for the degree to which either goal is met become more stringent. Meeting both goals simultaneously is even more difficult because techniques for achieving them individually conflict with achieving both simultaneously. Current software development techniques may not be adequate in many instances and, in practice, users demanding the highest performance end up rewriting their applications or each platform. This has lead many researchers to the conclusion that different program variants are required in different computational situations to achieve both high performance and portability.

Adaptive algorithms, sometimes referred to as polyalgorithms, are constructed such that they encapsulate a variety of methods to solve a single problem. The specific method applied to a given problem is determined at run time based upon the characteristics of the problem being solved, the machine being used, and so on. The software mechanism responsible for making this determination is called a *switching function*. This paper examines several alternative formulations for switching functions based on empirical performance modeling. The performance of each formulation is assessed in terms of accuracy and cost tradeoffs, as well as in terms of the performance insight provided by the functions themselves. Lessons derived from these analyses are then applied to the creation of a working adaptive algorithm for parallel computation.

Introduction

This research is focused on technologies that enable development of programs with characteristics that Alpern has termed the three P's: portability, high performance, and parallelism (Alpern and Carter 1994). More specifically, it is focused on enabling high performance, portable scientific and engineering applications for use in modern heterogeneous high performance computing (HPC) environments. These applications typically employ mathematically complex models and produce large amounts of data. High performance computers have long been employed by scientists and engineers to meet the computational challenge of producing accurate results from complex models within operational time constraints. In recent years, however, the computational resources in large computing environments are increasingly divided over several computers, rather than being centrally located on one monolithic HPC machine. In order to assemble sufficient resources to meet the objectives of a computation it may be necessary to utilize more than one system. This requires that different versions of the application be developed and maintained for each system used, or that a single version supporting each system be developed. In environments where only a single computer is used, the specifics of hardware and software configuration in high performance environments are often dynamic, and a single machine may undergo several incremental modifications throughout its installation. These modifications may alter the performance profile of the system, requiring application modifications to restore performance to expected levels. Resource decentralization and dynamic system configuration contribute to the need for application portability. Portability preserves existing code development investments, and enables research staff to focus on their research rather than the nuances of facilitating the computations themselves.

The importance of achieving high performance, portability, or both to the research community is indicated by the development of a tremendous array of software, software libraries, and standards over the years. The construction of software solutions to these problems has been complicated by the fact that programming for high performance usually requires adding machine-specific program constructs into an application, an approach that is simultaneously at odds with creating portable programs. The source code portability of applications between machines has been well served by efforts to standardize implementation languages. One may reasonably expect an application written in ANSI C to compile on all platforms of interest. However, the machine-specific constructs for high performance are dependent upon the particular features of a machine's hardware. This naturally hinders portability, and so portability of the implementation language, while useful, is clearly not sufficient for our goals.

One of the most common software design mechanisms employed for achieving high performance and portability is abstraction. This is usually accomplished via standardized interfaces to machine-dependent implementations. Numerical libraries starting with the BLAS (Lawson, *et al.* 1979)(Dongarra, *et al.* 1990)(Dongarra, *et al.* 1988) and proceeding through LINPACK, LAPACK (Anderson, *et al.* 1995), ScaLAPACK (Dongarra, van de Geijn, and Walker 1992), ScaLAPACK++ (Dongarra, Pozo, and Walker 1993), the Multicomputer Toolbox (Skjellum 1993), PETSc (Balay, Gropp, and Curfman 1998), PLAPACK (Alpatov, *et al.* 1997)(van de Geijn 1997), and many others, embody this approach. Utility libraries have also been constructed using the same principles, providing portability and performance across a multitude of platform-specific functions for similar functionality. Examples of this include message passing libraries on distributed memory parallel computers, such as MPI (Message Passing Interface Forum 1994) and PVM (Geist, *et al.* 1994). Standardization efforts such as OpenMP (Dagum and Menon 1998) have similar goals for shared memory parallel computers.

Limitations of Current Solutions

Each of these tools deliver a common set of functions over different computing platforms by using a standard Application Program Interface (API) that masks the vendor- and hardware-specific code necessary to achieve programmability with high performance, portability, or both. As such, they accomplish a portion of the tasks necessary to create portable parallel programs that also achieve high performance. These libraries typically support a single implementation for each functionality provided (such as matrix multiplication, LU decomposition, and so on). As demands on applications change and computation environments evolve over time, the performance of applications built using these libraries is likely to change, possibly for the worse. This is because traditional library design has focused upon selecting the *single* algorithm that performs best over the entire range of expected problems. In the traditional approach, the algorithm that best handles the problems expected to occur most often is implemented. Let us call the set of possible problems expected to occur most often the *design range* for an application. Performance may degrade when an algorithm selected for a specific design range is applied to problems outside that range. For some applications it may not be possible to define a design range that is sufficiently small to be adequately covered by a single algorithm. In cases where the designer can define an acceptable design range and select a suitable algorithm, this choice may not remain valid over time. The characteristics of input problems may evolve out of the design range as requirements continuously evolve (changing the nature of problems solved), hardware or system software is upgraded, or the application is ported to new architectures. Input characteristics may also evolve into a subset of the original design range, eliminating inputs at the *extrema* of the original range. This too may result in performance degradation if a different algorithm would perform more effectively on problems in the reduced design range.

Scientific and engineering applications often require high performance to meet simulation objectives, and a dynamic and diverse HPC environment requires that applications also be portable. Designing applications that achieve high performance on multiple machines is a significant challenge. Standardized implementation languages that facilitate portability are insufficient in view of the hardware-specific constructs that must be included in application code to achieve high performance. Furthermore, high level libraries that enable application designers to express functionality via a common interface while hiding hardware-optimized implementations from the designer are also insufficient. These libraries typically implement a single algorithm that may or may not perform within expectations on problems encountered by an application during production. A more flexible approach to application development is needed.

Adaptive Algorithms

Creating programs that are portable and achieve high performance is difficult, and as discussed above current approaches to addressing the challenges are inadequate in many instances. This has lead many researchers to the conclusion that different program variants are required in different computational situations to achieve high performance. Alpern and Carter (Alpern and Carter 1994) propose the need for different program variants in motivating the need for a generic model of computation. Other researchers propose that algorithms be written to include a variety of methods for solving a single problem, with a mechanism provided to select between the competitors at run time (Brewer 1995)(Demmel 1992)(Dongarra and Walker 1995)(Gunnels, *et al.* 1998)(Li, Skjellum, and Falgout 1997)(Rice 1968)(Sussman 1992)(van de Geijn and Watts 1997).

These approaches may be viewed as parts of a growing opinion that that adaptive algorithms have potential for creating algorithms that achieve portability and high performance on parallel computers.

Adaptive algorithms, sometimes referred to as polyalgorithms (Rice 1968) or poly-algorithms, encapsulate a variety of methods to solve a single problem. The specific method applied to a given problem is determined at run time based upon the parameters that influence performance: machine issues, application software issues, and instance issues. Adaptive algorithms can be thought of as containers for several related methods of solving a single problem along with a mechanism for selecting the best[1] algorithm for each problem. If enough algorithms are included in the algorithm set, chances are that at least one of them will perform well for a given problem on a given machine. As noted by Brewer (Brewer 1995) this approach simplifies code reuse and provides a framework for incorporating new algorithms. This approach also provides a basis for directing new algorithm research, and provides a mechanism for evaluating the effectiveness of those algorithms.

Current Approaches to Adaptive Algorithms

There is limited previous work directly examining the creation of switching functions for adaptive algorithms. However, a few numerical software packages are available that implement adaptive algorithms. The primary examples of these are PLAPACK (van de Geijn 1997), the Finite Difference Solver (FIDISOL), the Cartesian Arbitrary Domain Solver (CADSOL), and the Linear Solver (LINSOL). PLAPACK includes an adaptive algorithm for selecting among competing parallel matrix multiplication algorithms. The switching functions used are simple, comprised of only four rules that are each sensitive only to ratios of the matrix dimensions. These switching functions appear to be based on heuristics. Early results indicate a relatively low accuracy in predicting the most desirable algorithm (about 80% correct predictions in early tests[2]). This indicates that a more sophisticated approach may be needed.

An example of the approach typically taken in the other packages is found in LINSOL, which uses an adaptive algorithm for the iterative solution of linear systems by Krylov methods (Weiss, Haefner, and Schoenauer 1995). The adaptive algorithm uses three algorithms, each with different numerical characteristics. The first algorithm is very fast for diagonally dominant matrices, but may be unstable otherwise. The second algorithm converges more slowly, but the convergence criteria are somewhat relaxed. The final algorithm always converges very slowly, but is most likely to converge. The same approach is used in VECFEM3 for a subset of supported Krylov subspace solvers. Note that in these libraries the only criterion used in the switching function is convergence.

[1] Use of the word “best” in this context means “best among the available alternatives.” rather than “best among the universe of possible alternatives.”

[2] These tests were performed by the author in a set of preliminary investigations designed to precede the research and provide focus for future efforts.

There are many potential adaptation options for the Krylov methods, particularly with respect to communication patterns and data storage requirements, which are not included.

These packages are focused on delivering high performance by modifying computation patterns. There are other aspects on which algorithms may adapt, for example storage and communication patterns. SparsLinC supports adaptive algorithms for data storage. Sparse vector accumulation is implemented using an approach that transparently switches between three different representations of sparse vectors. Some implementations of MPI and the Bulk Synchronous Parallel library (Hill, *et al.* 1997) provide adaptive communication algorithms for global communication that are sensitive to the number of processors involved.

Also relevant are the ATLAS (Whaley and Dongarra 1998) and PHiPAC (Bilmes and others 1997) projects. Both projects seek to produce optimal code for computation kernels through code generation. Specifically, each seeks to optimize matrix multiplication on uniprocessors with multi-level memory hierarchy, although the research can easily be applied to other linear algebraic operations on the same types of machines. Both systems perform a suite of benchmarks upon installation to determine how the hardware features, such as cache size, etc., of the machine being evaluated impact algorithm performance. These projects are conceptually similar to the proposed research. However both ATLAS and PHiPAC focus on problems with a single adaptation dimension, matrix size, and are limited to uniprocessor problems. In addition, a single cost function, user execution time, is available for minimization. Perhaps most significant, however, is that both projects select the optimal algorithm at compile time.

The largest drawback for compile time approaches is that they are targeted for applications in which the performance-affecting features of a specific problem are known at compilation time. Thus the application must be recompiled whenever the characteristics of the machine or the problem change. This can be problematic if highly dynamic characteristics, such as the number and configuration of processors used on a problem, affects performance.

The most extensive treatment of adaptive algorithms for achieving high performance with portable sensitivity to machine and instance issues is given by Brewer (Brewer 1995). This work, which is based on research by Sussman (Sussman 1992), is the basis of the current research. The switching functions employed by Brewer are based upon models that are statistically generated from a database of empirical performance data. Brewer's linear regression approach is highly accurate (over 99% in published examples (Brewer 1995)), but is expensive to create, as the performance data upon which the regression is performed is gathered through an extensive benchmarking process. This process is equivalent to solving an optimization problem by exhaustive search of the parameter space. The cost of populating the instance space for this type of problem rapidly becomes large as the number of parameters increases, or when any one dimension is particularly large. Brewer reports running several tens of thousands of benchmarks in order to produce a single switching function. A central theme of the current research will be to examine alternatives to the straightforward linear regression approach and the accuracy/cost tradeoffs inherent in these alternative solutions.

Current Research

The current research examines adaptive algorithms as a mechanism for creating numerical software that achieves both high performance and portability. The research program is long term and is ongoing. The two aspects of adaptive algorithms proposed for study in this research are switching functions and cost models. Switching functions are the software mechanism by which the adaptive algorithm selects the best available implementation for the current computational system[3], a process also referred to as variant selection (where variant is short for algorithm variant). Work by previous researchers discussed above has concentrated on a single switching function technology with a high associated cost. A central theme of this research is to examine alternatives to this approach and the accuracy/cost tradeoffs inherent in those alternatives. The complete program of research, described elsewhere, includes study of switching functions developed from analytical performance models and geometric methods based on adaptive benchmarking. The results discussed in this paper focus on the development of switching functions based on the results of traditional data mining methods.

The data mining methods as a class are essentially techniques for automatically determining how observed characteristics of a problem are related to a given outcome. As such, they rely on the training data to contain accurate information about transitions between possible outcomes. The methods are particularly useful for problems that have large numbers of features whose relationship to outcome is not clearly understood. In this research these methods will be applied to performance measurements of algorithm variants in an effort to identify expressions for predicting which variant is best for a particular problem. Note that the key to the effectiveness of these methods is that they are often robust in the face of data not available when they were originally trained. We hope to show that this feature extends into the performance domain so that accurate predictions may be made for all problems in a given range from relatively few measurements. This is important because the type of extensive benchmarking performed in previous work (Brewer 1995) is expensive and often impractical in supercomputing environments.

Switching Functions and Data Mining

There are two general ways to formulate a problem for solution by data mining techniques. Problems may be formulated as either *classification* or *regression* problems. In data mining the characteristics of a particular problem are called *features* or *input features*. For classification problems the input features are used to predict which of the possible output states is the most likely outcome. For this type of problem there is more than one possible outcome; each outcome corresponds to a *class*. The inputs may be continuous or discrete features. The outputs are discrete – the predicted outcome is a given class (one), or it is not that class (zero). As an example, consider the classic loan assessment problem from financial management. The problem is to predict, given a set

[3] A *computational system* is the combination of hardware (a platform), system software, and the specific problem being solved.

of features, whether a potential customer will default on or repay a loan. The feature set could be any relevant data: current debt, whether the customer has defaulted in the past, present income, whether the customer owns a home, and so on. There is a single output which takes the value one (loan granted) or zero (loan denied).

The other way to formulate a problem is as a regression, or continuous output, problem[4]. A regression problem is one in which the input features predict the value of a single continuous output variable. Both inputs and outputs may be continuous; the inputs may also be discrete. For example, reconsider the loan assessment problem. This problem may be reformulated as a regression problem if, given the same input features, we desire a real-valued estimate of the default risk. The methods can then be trained to produce a quantitative indicator of risk expressed as a number between zero (no risk) and one (certain default).

The loan assessment problem may be posed as either a classification or a regression problem. Many problems lend themselves to both formulations. The switching function problem in the proposed research may be posed either way. The regression formulation would take the input features and provide a real-valued estimate of the performance for each of the available variants. An extra layer of logic would then be added to select the algorithm with the highest predicted performance from the variant set, and apply this algorithm to the problem. For this approach a separate model would be constructed from training data for each of the variants. The classification formulation would take the same set of inputs (possibly), but rather than producing a single continuous output it would provide *m* discrete outputs, one for each algorithm variant. The output value for each algorithm would be zero except for the algorithm that is predicted to be most desirable for the current problem. Figure 1 and

Figure 2 conceptually illustrate the differences between these two approaches.

The application of data mining methods to benchmark data represents a departure from both the traditional applications of data mining methods and the accepted computational engineering approaches. The novelty of the application suggests that it is wise to expend a small amount of initial effort in a pilot exploration to establish the feasibility of the full course of research. This paper summarizes the results of part of that pilot study.

Computational Infrastructure

The switching functions explored in this research are targeted for selecting among variants for numerical computation on parallel distributed memory HPC platforms. PLAPACK (van de Geijn 1997) has been selected as the software infrastructure for this research for several reasons. First, it supports manipulation of high level mathematical objects while maintaining performance at least equivalent to ScaLAPACK. Second, the

[4] Problems in this category are referred to as regression problems, although this terminology is not meant to imply that the standard linear regression methods from statistics are applied in every instance.

Physically Based Matrix Distribution (PBMD) approach to data distribution is intriguing, and appears to have certain advantages for minimizing the complexity of parallel algorithms. Finally, PLAPACK supports a rudimentary matrix multiplication adaptive algorithm to which the results of the current research can be compared.

Adaptive Matrix Multiplication in PLAPACK

It would be advantageous to reuse the matrix multiplication code in PLAPACK as the foundation for the matrix multiplication portion of the proposed research. Since the emphasis is on mechanisms for selecting among algorithm variants, it is not necessary to have a fully functional adaptive algorithm that addresses the complete range of potential problems from an application domain. Rather, all that is necessary is to assemble some reasonable collection of multiplication algorithms that is sufficient to test the switching technologies being investigated.

The matrix multiplication variants in PLAPACK are developed by studying the many ways to express matrix multiplication using the BLAS. The most common method is to simply call the **gemm** family of routines. However, on many architectures the performance of this routine is highly influenced by the shape of the matrices being multiplied. Let us consider the multiplication `C=AB`, where `C` is `m` by `n`, `A` is `m` by `k`, and `B` is `k` by `n`. As the sizes of the three dimensions vary, the shapes of the matrices being multiplied varies. For example, if dimension `k` becomes small then `A` becomes a column vector and `B` a row vector. The most desirable BLAS routine for multiplying these vectors is the **ger** family. Letting each of the dimensions shrink to one and creating the corresponding eight matrix shapes one can create a complete table of shapes and the most desirable BLAS operation for the multiplication of those shapes. This is the approach taken by PLAPACK in its shape-adaptive matrix multiplication algorithm. A complete description of the different operand shapes and the determination of appropriate BLAS routines are given in (Gunnels, *et al.* 1998). The key to choosing which variant to use for a specific problem lies in quantifying the adjectives “small” and “large.” This is the job of the switching functions.

The code segment below shows the PLAPACK switching functions for parallel matrix multiplication. In this code segment `m`, `n`, and `k` are the matrix dimensions; `nb` is the blocking factor; and `nprocrows` is the number of rows in the virtual processor topology[5].

```
If [ (10*m<k) and (10*n<k) and
      m < (nb*nprocrows/2))and
      n < (nb*nprocrows/2))    ]

then variant 4

else if [ (5*n<k and 5*n<m)or
          (5*m<k and 5*n<k)    ]
```

[5] The processors are arranged into a `nprocrows` X `nproccols` grid.

```
then variant 2
else if [5*m<k and 5*k<n]
then variant 3
else     variant 1
```

Note that this switching function only includes four of the possible eight matrix multiplication variants created by varying the index sizes as discussed in the preceding section. These four correspond to matrix dimensions shown in Table 1. The PLAPACK library is distributed with six implementations; the remaining two (m, n, k small and m, n large, k small) are not included. Independent experiments on a wide range of matrix sizes indicates that the two implementations not included in the switching function are never most desirable on the measured architectures, despite theoretical indications to the contrary. This is a result of the mechanics of actual hardware implementations and data movement through the memory hierarchy. Measurements have shown that algorithm four is only the most desirable variant when m, n, and k are each very small. In the benchmarks for this study algorithm four never wins.

Measurements

In order to derive a qualitative understanding of how the proposed techniques may fare in a complete application, a comparative study has been designed. This study applies each of the proposed techniques for generating switching functions to sets of algorithm performance data in order to evaluate how switching functions generated with each of the methods perform. These sets of data are called training sets. The switching functions are then used to make predictions of the most desirable algorithm for a set of new problems. These problems were completely hidden from all methods during the development of the switching functions. Data used for evaluating the switching functions are called test sets.

The switching functions built in this preliminary study are for parallel matrix multiplication on an IBM SP. The configuration of the SP used for the data mining study is shown in Table 2 and Table 3. PLAPACK version 1.2 is used for this study. PLAPACK requires LAPACK for some of its operations. LAPACK is not installed on this machine; LAPACK functionality was provided by building the kernels that accompany the PLAPACK distribution. PLAPACK also requires the BLAS for all local computations; the vendor-optimized BLAS library was used. The matrices multiplied are m by k and k by n, producing and m by n product matrix. Each dimension is allowed to vary in a triple-loop. Performance in MFLOPS is measured for each algorithm variant at each problem size. The reported performance for each variant is the average of three executions per problem on a production machine. In order to ensure that the data are not biased by cache effects the cache is flushed between each individual measurement. For this study the four matrix multiplication variants supplied with PLAPACK as discussed above are used.

Initially the switching function problem will be expressed as a classification problem. Intuitively this is the "right" way to express this problem. Practically, however, an objective evaluation of both the continuous and discrete formulations will need to be made before a final decision is made. The methods may select among four algorithm variants, and thus there are four output classes. There are six input features. The matrix dimensions `m`, `n`, and `k` are considered, as are the ratios of these dimensions (`m/n`, `m/k`, `n/k`). The ratios were included to provide the opportunity for the data mining methods to re-create the PLAPACK switching functions. This might happen if, for example, the PLAPACK switching functions did turn out to be a good model for the phenomena in the parameter space. Note that this is a small subset of the factors the influence the performance of parallel matrix multiplication algorithms. The number of processors, shape of the process topology, distribution and algorithm blocking factors, and others are being considered in ongoing research. These results are not discussed in this paper.

One training set and two test sets were generated for this evaluation.

Table 4 lists the exact dimensions used in generating each set of performance data. In order to create the training sets the performance of each variant on each problem has to be measured. These measurements were taken on four processors with a square processor topology (2x2). Benchmarks for the test sets are also performed in order to check the accuracy of the switching function predictions. In order to assess how well a switching function trained with data generated on one processor configuration (2x2) performs in predictions for different processor configurations the test data were also benchmarked on 3x3 and 4x4 processors.

The prevalence of each class for each data on the various processor configurations is shown in Table 6 through Table 13. The remaining characteristics are described below:

- Training Set 1
  There are thirteen possible index values. Letting each index assume each value results in 2,197 separate problems. Each variant is measured for each problem; for four variants, this creates 8,788 separate measurements

- Test Set 1
  There are nine possible index values. These values were selected by choosing sample points between samples in Test Set 1 and perturbing the indices by a random number between –10 and 10. Each of the indices in this test set is contained with the bounds of the training sets. Therefore, this set is also called the Inside Test Set. Note that none of these index values appears in either of the training sets, so this is a clean test set – each problem size is completely new. Letting each index assume each value results in 729 separate problems. Each variant is measured for each problem; for four variants, this creates 2916 separate measurements.

- Test Set 2
  There are five possible index values. These values were selected by choosing sample points outside the samples in Training Set 1. Where the Inside Test Set measures performance of the switching functions on problems that are within the range of the original training data, Test Set 2 will measure the ability of the switching functions to extrapolate to new problems. This set is also called the Outside Test Set, and is a clean test set. Letting each index

assume each value results in 125 separate problems, and there are 500 separate measurements.

For measurements taken on the 3x3 and 4x4 processor configurations the size of each problem remains constant on a per processor basis. The amount of work performed by each processor is held constant by scaling each value of `m`, `n`, and `k` by the square root of the number of processors. During training of the methods, however, the data sets express results in terms of the base value of `m`, `n`, and `k` for each processor configuration. Thus, if a 50 by 50 problem is run on 2x2 processors, the benchmark is executed as a 200x200 element problem, but results are recorded in the training and test sets as the original 50x50 size. One may think of the methods being trained to respond to the amount of work per processor, not just the global problem size. This is done so that direct comparisons can be made between the results of data mining methods applied on different processor configurations.

Data Mining Methods

In this pilot study, we will investigate the performance of switching functions generated from relationships identified by logic, math, and distance data mining methods. Specifically decision trees, linear discriminant analysis (LDA), and *k*-Nearest Neighbors (*k*-NN) methods will be evaluated. The complete program of research includes other methods as well. Accompanying the data mining methods are pre-processing methods for the data themselves. These methods can enhance predictive performance or reduce training time by changing the training data set in a variety of ways. We will apply feature and case reduction methods. Most of the methods employed in this study have parameters that can be tuned for optimal performance. In this preliminary study, these parameters are fixed. Thus, although the results of this study are not necessary optimal, they will allow the relative effectiveness of the different methods to be determined.

Only a rudimentary discussion of the data mining and pre-processing methods will be provided – the reader seeking information that is more detailed will find an excellent overview in (Weiss and Indurkhya 1998).

Pre-Processing Methods

Pre-processing methods generally fall into three categories: feature, value, and case reduction methods. Feature reduction methods apply transformations to the data to eliminate one or more input features from the training set. For example, the statistical significance of each feature relative to the output classes may be computed. Features not determined to be significantly different across the classes are not useful for prediction, and may therefore be eliminated. Feature reduction is most often useful for math and distance methods because the do not have this capability built in. This study will employ tree reduction over the feature set for determining which features may be eliminated. In this approach a decision tree solution is induced from the training set exactly as would be done if the tree method was applied to the training data to produce a predictive solution. However, for feature reduction we are not concerned with the structure of the tree or its relationships, but only whether a feature appears at all in the decision-making process. Features that do not appear in the tree are eliminated from the training set.

Value reduction methods are usually applied to data sets that have input features that take on many (thousands) of values. These methods reduce the total number of unique values by clustering or assigning bins into which the values are grouped. This can result in considerable computational savings for methods that rely on repeatedly sorting the features. Value reduction is not explored in this study. Three of the six input features are integers, and so do not represent a value-rich set. The remaining three are ratios of the first three input features and so take on relatively few distinct real values.

Where feature reduction eliminated features from all training cases, case reduction eliminates individual cases from the training set. One may think of the training data as being organized in a spreadsheet. Then features are the columns and the cases, organized in rows, are collections of features. Feature reduction thus eliminates columns, while case reduction eliminates rows. Random case subsampling is the most common case reduction methods. In this approach a specified percentage (10%, 60%, etc.) of the cases are chosen randomly and removed before training begins. This approach is most often used on large data sets to create the training and test sets needed for critical evaluation.

Random case subsampling does not make sense in the present research as groups of cases are related to one another. Case subsampling will be applied to all data mining methods by selecting specific m values and removing the `n*k` cases associated with each `m` index. This can be repeated many times, retraining the data mining methods after each extraction. If prediction accuracy is monitored as the number of cases is reduced, it may be possible to determine whether a smaller number of benchmarks could have been performed while maintaining satisfactory performance in the switching functions. This approach is referred to as incremental case analysis. Rather than performing an exhaustive incremental analysis, only selected increments will be tested. In this study three subsets of Training Set 1 are used. The first is created by eliminating every third sample. The second is created by eliminating every other sample. The final subset is created by keeping every third sample. Table 5 gives the exact dimensions for these cases.

Data Mining Methods

Data mining methods may be grouped into three categories: math, logic, and distance methods. In this study the math method employed is linear discriminant analysis (LDA) – results from training with neural networks are being developed and may be presented at the conference. LDA is the discrete version of the classic linear statistical regression, while neural networks represent a non-linear approach to classification. Tree reduction will be applied to the features for the LDA. Incremental case subsampling as described above will be applied to this method as well (in fact, case subsampling will be applied to all methods).

Decision tree solutions from the logic methods will be computed on the training data as well. Feature reduction is automatic with this method. Solutions of varying complexity may be obtained from a single decision tree by pruning subtrees that have a predictive value below a certain significance threshold. Several such prunings will be performed for each decision tree for comparison. Significance levels of s=1.6 and 2.0 will be studied.

The distance solution investigated in this study is *k*-Nearest Neighbors, or *k*-NN. This method makes a prediction by selecting among the *k* cases in the training set, for which the correct solution is known, that are closest to the current problem. Closeness must be defined by an appropriate distance metric – the metric employed in this research is the Euclidean distance (or its *n*-dimensional analogue). Solutions with *k* equal to 1, 5, and 11 will be evaluated.

The performance of the methods is expressed in terms of a classification error rate. This rate is computed by dividing the number of misclassifications by the total number of switching function evaluations. In each of the tables the error rate is expressed as a percentage.

Results

Before we get started its interesting to know how the PLAPACK switching functions behave on the Inside and Outside Test sets for comparison to the data mining methods. Table 14 shows error rates for the PLAPACK switching functions. Performance is measured against benchmarks on 2x2, 3x3, and 4x4 processor configurations. Note in all cases that the error rate is high, especially when compared to the error rates of less than 1% obtained by Brewer (Brewer 1995).

Decision Tree

Three decision trees were induced from the training set: the default tree, and trees pruned at significance levels of 1.6 and 2. These particular values were selected based on earlier research indicating these significant levels often yield effective solutions. Table 15 shows the performance of these decision trees. Decision tree solutions are expressed as binary trees where each node is either a condition or a decision. Tree methods recursively partition the feature space by features and feature values. Features closer to the root of the tree are the most significant predictors of outcome. The root node of the default tree tests whether feature 5, the ratio m/k, is greater than .523. If so, the processing continues in the left subtree. If not, processing continues in the right subtree. Subtrees terminate with leaf nodes when enough is known about a particular problem to predict which algorithm is the best. The default decision tree has 303 nodes; the decision tree pruned for significance levels of 1.6 and 2.0 have 99 and 51 nodes, respectively. The reduction in nodes creates an accompanying reduction in complexity of the switching functions.

Pruning may also improve predictive performance by removing subtrees that are too specific – a condition known as over-training. Over-training occurs when data mining methods are provided with enough information that they produce specialized solutions that are highly accurate on the training data, but produce diminished accuracy on new cases. The results in Table 15 demonstrate that pruning can be helpful. The tree pruned to a significance of 2.0 provides the most accurate results on the Inside Test. Studying the error rates in this table indicates that there may be a decreasing error trend beyond the cases reported in the table. This is not the case – further analysis at significance levels of 2.5 and 4 shows the error increasing once again for all processor configurations so 2.0 is a good choice.

Note that this table also contains columns for performance on 3x3 and 4x4 CPU configurations. These measurements show that performance deteriorates when switching functions trained from benchmark data generated on one processor configuration (2x2 in this case) is used to make predictions of performance on other configurations. Note that not only does the misclassification rate increase on 3x3 processors, but that this rate continues to increase as we progress to 4x4 processors. Recall that the PLAPACK switching functions are not sensitive to the number of processors used or their topology. This result provides some indication that this approach is not sufficient if switching functions of a fixed accuracy on all processor configurations are desired.

Table 16 provides results for decision trees trained by subsampling the original training set, Training Set 1. Case subsampling can improve computational performance by reducing the number of cases that need to be considered in inducing a solution. This is especially important for large data sets – our data sets are not considered large by data mining standards, but the data are expensive to generate and so minimizing the size of the training set is important. Case subsampling may also improve predictive performance by preventing over-training. Table 16 shows the most accurate predictions are derived from a switching function trained on the second subsampled data set and pruned to a significance of 1.6.

k-Nearest Neighbors

Three distance solutions were created using the *k*-NN method for values of k equal to 1, 5, and 11. These values were selected based on earlier research indicating that these were reasonable values. As with decision trees, *k*-NN based switching functions were created from Training Set 1 as well as from case subsampled versions of this data set. Feature reduction by tree selection was also applied to the training data. As discussed above, where case subsampling removes rows of a spreadsheet, feature reduction removes columns. Which features to remove is determined in this case by first determining which features the tree method includes in its solutions. If a feature appears anywhere in a decision tree it is left in the training spreadsheet. For this study three trees were examined: the default tree, the tree pruned at a significance of 1.6, and the tree pruned at a significance of 2.0. The default tree contains all features, the s=1.6 tree does not include matrix dimension m, and the s=2.0 tree does not include dimensions m or k. The ratios of dimensions appear in each case. This indicates that the ratios are more statistically significant than the dimensions, and provides some justification for the PLAPACK decision to only include ratios in its switching. However, this decision is not completely justified, as the error rates for the PLAPACK switching functions are several times higher than the *k*-NN predictions.

Table 17 to Table 19 show the performance of the nearest neighbor methods. As with the decision tree method, the performance of 2x2 processor trained switching functions on benchmark data from different configurations is also shown, with similar results. For 2x2 switching functions measured on 2x2 data, feature reduction always increases the error rate. This observation reinforces the idea that it is not sufficient to eliminate the matrix dimensions from the switching function formulation. The best performance on both data sets is obtained by the 5-NN switching function with no feature reduction.

Table 20 shows the results for the nearest neighbor method with case subsampling. In all tests case subsampling increases the prediction error rates.

Linear Discriminant Analysis

Linear Discriminant Analysis, or LDA, was the final data mining method evaluated for its switching potential. Table 21 shows the performance of this method on the standard data sets, for both the 2x2 and other processor configurations. Performance with and without feature reduction is shown. Feature reduction is accomplished using tree selection of relevant features, as was done for the nearest neighbor method discussed above. As in the other cases the prediction accuracy continues to degrade the further the processor configuration for which predictions are made differs from the training configuration (2x2 in this case). The performance of the method without feature reduction is significantly better than with feature reduction.

Table 22 shows results on the 2x2 test sets with case subsampling. This table reveals that case subsampling has little effect on prediction accuracy for these test sets.

Conclusions

This preliminary investigation examined the application of data mining methods to benchmark data for creating switching functions for parallel matrix multiplication. Despite the preliminary nature of this study, much useful information has been obtained. First, the accuracy of the data mining methods is significantly better than the default PLAPACK switching functions in this study. This provides adequate motivation for commencing the complete research program. The switching functions produced by the data mining methods achieve 90-95% accuracy. This is lower than the 99% accuracy reported by Brewer (Brewer 1995). Brewer achieved this accuracy using linear regression models of performance created from benchmark a database of tens of thousands of measurements. It is important to note that this study achieved 95% prediction accuracy (using a decision tree) from a database of just over one thousand measurements (343 benchmarks times four algorithms per benchmark). In production environments where it may not be possible to set aside days of dedicated computation for calibrating numerical software switching functions this reduction in the benchmarking requirement is significant. Furthermore, it is likely that the application of additional standard techniques from data mining may further improve this prediction rate, possibly even while reducing the benchmarking requirement further.

Of the methods reported in this paper decision trees and nearest neighbor methods appear to have the most potential. Both of these methods produce switching functions that are easily represented in programs as a set of nested `IF` or case statements, and so are rapidly evaluated. Both methods also have negligible training times. This may be important for numerical software to retrain itself during production deployment as new performance data is gathered as a result of user utilization.

Two pre-processing methods were evaluated – feature reduction and case subsampling. Case subsampling was effective for decision trees. This is less important for the slight improvement in accuracy than because it demonstrates that accuracy can be maintained while significantly reducing the quantity of training benchmark samples that are required (nearly 50%). Table 23 summarizes the costs of obtaining the training and test data for all cases. The degree to which this applies when the switching functions are generalized to support a broader range of problem sizes remains to be seen. Neither case subsampling or feature reduction was effective on the nearest neighbor methods. Post-processing the decision trees by pruning also proved effective. The benchmarked significance levels were supplemented by additional tests that show that significance levels of 1.6 and 2.0 appear to have the most potential.

Table 24 contains the prediction penalty analysis for PLAPACK, LDA, 5-NN, and the decision tree pruned to a significance of s=2.0. The prediction penalty is computed by finding the difference in performance associated with using the predicted algorithm over the correct algorithm for that problem. Thus the prediction penalty is the cost of making a prediction error. The performance penalty is expressed by computing the difference between the correct and predicted algorithms. This difference is then divided by the performance of the correct algorithm and expressed as a percentage. The analysis in Table 24 is done for the Inside Test Set, and shows that both the 5-NN and decision tree methods have penalties that are three times less than those of PLAPACK, on average. Table 25 repeats the analysis for the decision tree pruned at s=1.6 to illustrate that, although training costs can be reduced while maintaining prediction error the average cost of those errors may (and in fact will probably) increase.

As a final note recall that, in order to reduce variability in the benchmark data, the average of three runs was used in creating the training sets. The three runs were made immediately following one another. However, for cases where a benchmark had to be repeated substantially later (a day or more), differences in average performance for the same problem were observed to be as high as 9%. This suggests that future efforts should average results gathered over a broader time frame to ensure the results are more representative of what a user will actually encounter.

Acknowledgements

The author would like to gratefully acknowledge the insightful guidance of Prof. Geoffrey Fox (Northeast Parallel Architectures Center at Syracuse University) and Dr. Louis Turcotte (Information Technology Laboratory, USAE Waterways Experiment Station) in formulation of this research, as well as the energizing early discussions with Prof. Tony Skjellum (Mississippi State University).

The work described and the resulting data presented herein, unless otherwise noted, were obtained from research conducted as part of the DoD High Performance Computing Modernization Program at the United States Army Corps of Engineers Waterways Experiment Station. Permission was granted by the Chief of Engineers to publish this information.

Bibliography

Alpern, B., L. Carter, E. Feig, and T. Selker. "The Uniform Memory Hierarchy Model of Computation." *Algorithmica* 12, 2-3 (August-September 1994).

Alpatov, P., G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, and R. van de Geijn. "PLAPACK: Parallel Linear Algebra Package." *Proceedings of the SIAM Parallel Processing Conference,* 1997.

Anderson, E., Z. Bai, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. 2nd. SIAM, 1995.

Balay, S., W. Gropp, and L. Curfman. *PETSc 2.0 Users' Manual*. Http://www.msc.anl.gov/petsc/docs/manual/manual.html: Accessed March 1998.

Bilmes, J., K. Asanovic, C. W. Chin, and J. Demmel. 1997. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ansi c coding methodology. *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, ACM SIGARC. July.

Brewer, E. A. "High-Level Optimization Via Automated Statistical Modeling." *Proceedings of Principles and Practice of Parallel Programming*, 1995. Pages 80-91.

Dagum, L., and R. Menon. "OpenMP: an Industry-Standard API for Shared-Memory Programming." *IEEE Computational Science and Engineering* 5, 1 (January/March 1998).

Demmel, J. *Trading Off Parallelism and Numerical Stability*, University of Tennessee, 1992. Technical Report UT-CS-92-179; LAPACK Working Note 52.

Dongarra, J., Du Croz, J., S. Hammarling, and R. Hanson. "An Extended Set of FORTRAN Basic Linear Algebra Subprograms." *Transactions on Mathematical Software* 14, 1 (1988): 1-17.

Dongarra, J., Du Croz, J., S. Hammarling, and R. Hanson. "A Set of Level 3 Basic Linear Algebra Subprograms." *Transactions on Mathematical Software* 16, 1 (1990): 1-16.

Dongarra, J., R. van de Geijn, and D. Walker. "A Look at Scalable Dense Linear Algebra Libraries." *Proceedings of the 1992 Scalable High Performance Computing Conference*, J. H. Saltz., IEEE Press. 1992. Also appears as LAPACK Working Note 43, University of Tennessee Technical Report UT-CS-92-155, May 1992.

Dongarra, J., R. Pozo, and D. Walker. "An Object-Oriented Design for High Performance Linear Algebra on Distributed Memory Architectures." *Proceedings of the Object-Oriented Numerics Conference*, 1993.

Dongarra, J., and D. Walker. "Software Libraries for Linear Algebra Computations on High Performance Computers." *SIAM Review* 37, 2 (June 1995): 151-180.

Geist, A., A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, a Users' Guide*. MIT Press, 1994.

Gunnels, J., C. Lin, G. Morrow, and R. van de Geijn. "Analysis of a Class of Parallel Matrix Multiplication Algorithms." *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98),* 1998. Pages 110-116.

Hill, J., W. F. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. *BSPlib: The BSP Programming Library*, Oxford University Computing Laboratory, May 1997. Technical Report PRG-TR-29-9.

Lawson, C., R. Hanson, D. Kincaid, and F. Krogh. "Basic Linear Algebra Subprograms for FORTRAN Usage." *Transactions on Mathematical Software* 5, 3 (1979): 308-323.

Li, J., A. Skjellum, and R. Falgout. "A Poly-Algorithm for Parallel Dense Matrix Multiplication on Two-Dimensional Process Grid Topologies." *Concurrency: Practice and Experience* 9, 5 (1997).

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard,* Computer Science Department, University of Tennessee, Knoxville, TN, May 5, 1994. Technical Report CS-94-230. Also appears in the International Journal of Supercomputing Applications, Volume 8, Number 3/4, 1994.

Rice, J. R. "On the Construction of Poly-Algorithms for Automatic Numerical Analysis." In *Interactive Systems for Experimental Applied Mathematics*, M. Klerer and J. Reinfelds. 301-313. Academic Press, 1968.

Skjellum, A. "The Multicomputer Toolbox: Current and Future Directions." *Proceedings of the Scalable Parallel Libraries Conference*, A. Skjellum and D. S. Reese., IEEE Computer Society Press. October 1993. Pages 94-103.

Sussman, A. "Model-Driven Mapping Onto Distributed Memory Parallel Computers." *Proceedings of Supercomputing '92,* Minneapolis, MN, IEEE Computer Society Press. August 1992. Pages 818-829.

van de Geijn, R. 1997. *Using PLAPACK*. MIT Press.

van de Geijn, R., and J. Watts. "SUMMA: Scalable Universal Matrix Multiplication Algorithm." *Concurrency: Practice and Experience* 9, 4 (1997): 255-274.

Weiss, R., H. Haefner, and W. Schoenauer. *LINSOL (LINear SOLver) - Description and User's Guide for the Parallelized Version*, University of Karlsruhe Computing Center, 1995. Technical Report 61/95.

Whaley, R. C., and J. Dongarra. 1998. Automatically tuned linear algebra software. *Proccedings of Supercomputing '98,* Orlando, FL, IEEE Computer Society Press. November 7-13.

Figure & Captions

Figure 1 Continuous switching function formulation.

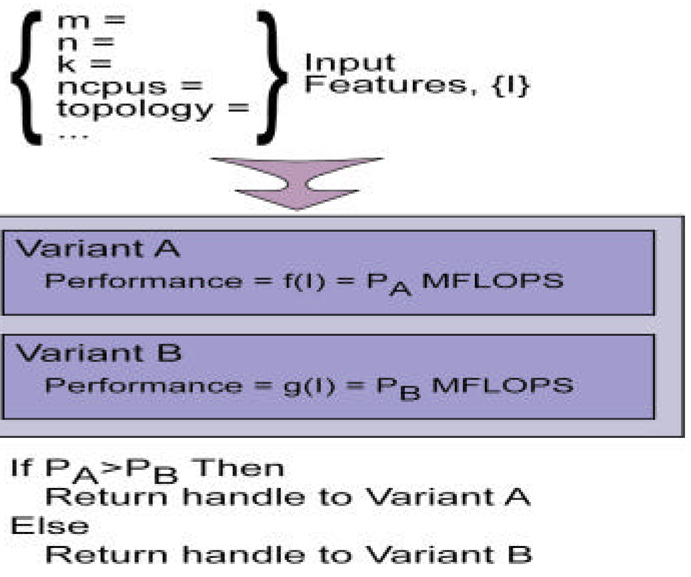
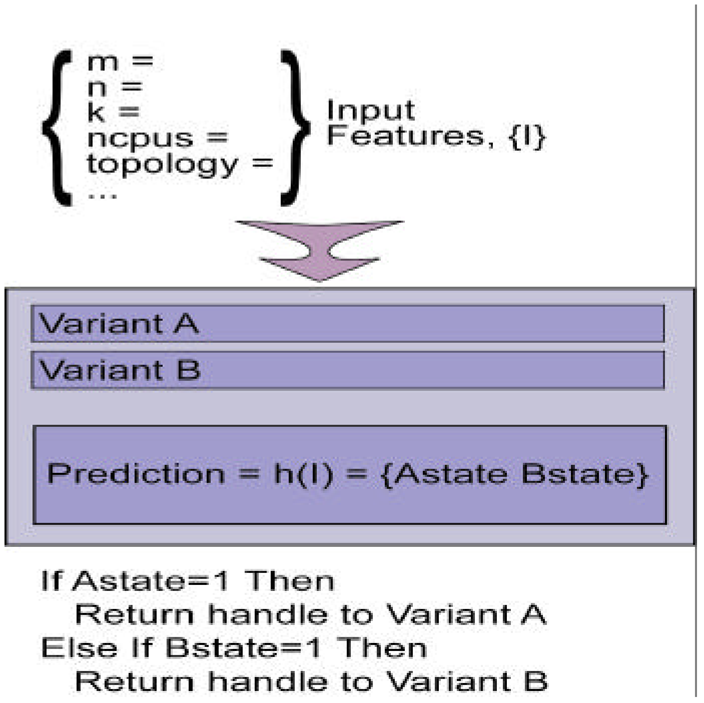

m =
n =
k =
ncpus =
topology =
...
Input Features, {I}
Variant A
Performance = f(I) = P_A MFLOPS
Variant B
Performance = g(I) = P_B MFLOPS
If $P_A > P_B$ Then
Return handle to Variant A
Else
Return handle to Variant B

Figure 2 Discrete switching function formulation.



Tables

Table 1 Algorithm variants for the PLAPACK adaptive matrix multiplication algorithm.

VARIANT	M	N	K
1	Large	Large	Large
2	Large	Small	Large
3	Small	Large	Large
4	Small	Small	Large

Table 2 IBM SP hardware configuration for the data mining study.

MACHINE	IBM SP (OSPREY)
Processors	256 P2SC CPUs (232 compute)
	135 MHz
Memory[*]	512 MB (80)
	1 GB (96)
	2 GB (56)

Table 3 IBM SP software configuration for the data mining study.

SOFTWARE	VERSION
Operating System	AIX 4.3.2.x
ESSL	3.1.0.1
Parallel ESSL	2.1.0.0
PBS	CM 1.1.x
POE	2.3.0.10
Compilers	
C	3.6.6.0
FORTRAN	6.1.0.0

[*] Memory is reported in Size (Num) groups, where Size is the amount of memory and Num is the number of processors with that amount of memory.

Table 4 Values of <m,n,k> for the three size classes in the data mining study.

DATA SET	<M,N,K>
Training Set 1	50,75,100,150,200,250,300,350,400,450, 500,550,600
Inside Test	60,83,130,174,232,319,373,418,530
Outside Test	40,45,650,700,750

Table 5 Values of <m,n,k> for the three training subsets.

DATA SET	<M,N,K>	REDUCTION (%)
Original Training Set 1	50,75,100,150,200,250, 300, 350,400,450,500,550,600	0
Subset 1	50,75,150,200,300,350, 450,500,600	30.7
Subset 2	50,100,200,300,400,500,600	46.1
Subset 3	50,150,300,450,600	61.5

Table 6 Class prevalence for Training Set 1 on 2x2 processors.

CLASS	CASES	PREVALENCE
1	1418	.645
2	322	.147
3	457	.208
4	0	0.0

Table 7 Class prevalence for the Inside Test Set (Test Set 1) on 2x2 processors.

CLASS	CASES	PREVALENCE
1	492	.675
2	96	.132
3	141	.193
4	0	0.0

Table 8 Class prevalence for the Outside Test Set (Test Set 2) on 2x2 processors.

CLASS	CASES	PREVALENCE
1	76	.608
2	20	.160
3	29	.232
4	0	0.0

Table 9 Class prevalence for the Inside Test Set (Test Set 1) on 4x4 processors.

CLASS	CASES	PREVALENCE
1	473	.649
2	116	.159
3	140	.192
4	0	0.0

Table 10 Class prevalence for the Outside Test Set (Test Set 2) on 4x4 processors.

CLASS	CASES	PREVALENCE
1	72	.576
2	25	.20
3	28	.224
4	0	0.0

Table 11 Class prevalence for the Inside Test Set (Test Set 1) on 3x3 processors.

CLASS	CASES	PREVALENCE
1	458	.628
2	113	.155
3	158	.217
4	0	0.0

Table 12 Class prevalence for the Outside Test Set (Test Set 2) on 3x3 processors.

CLASS	CASES	PREVALENCE
1	70	.560
2	26	.208
3	29	.232
4	0	0.0

Table 13 Class prevalence for subsampling of Test Set 1 on 2x2 processors.

	SUBSAMPLE 1		SUBSAMPLE 2		SUBSAMPLE 3	
CLASS	CASES	PREV.	CASES	PREV.	CASES	PREV.
1	958	.63	748	.632	524	.620
2	213	.14	162	.137	112	.133
3	350	.23	273	.231	209	.247
4	0	0.0	0	0.0	0	0.0

Table 14 PLAPACK switching function error rates (%) for Test Set 1 and 2.

	TEST SET	
CPUS	IN	OUT
2x2	27.3	12.0
3x3	32.0	18.4
4x4	29.9	16.8

Table 15 Decision tree error rates (%) for 2x2 processor switching functions.

	2X2 CPU		3X3 CPU		4X4 CPU	
SIG.	IN	OUT	IN	OUT	IN	OUT
1.0	7.3	2.4	9.6	10.4	13.4	9.6
1.6	6.2	2.4	8.4	10.4	12.5	9.6
2.0	5.9	2.4	8.0	10.4	12.5	9.6

Table 16 Decision tree error rates (%) for 2x2 processor switching functions with case subsampling on Training Set 1.

	FULL		SUB 1		SUB 2		SUB 3	
SIG.	IN	OUT	IN	OUT	IN	OUT	IN	OUT
1.0	7.3	2.4	7.5	2.4	6.9	2.4	9.5	2.4
1.6	6.2	2.4	6.2	2.4	5.6	2.4	7.3	2.4
2.0	5.9	2.4	6.9	2.4	6.3	2.4	7.3	2.4

Table 17 1-NN error rates (%) for 2x2 processor switching functions.

	2X2 CPU		3X3 CPU		4X4 CPU	
FEATURE REDUCTION	IN	OUT	IN	OUT	IN	OUT
None	7.8	2.4	10.2	10.4	11.5	9.6
1.6	6.7	3.2	10.0	9.6	12.2	8.8
2.0	8.0	3.2	10.4	9.6	14.8	8.8

Table 18 5-NN error rates (%) for 2x2 processor switching functions.

	2X2 CPU		3X3 CPU		4X4 CPU	
FEATURE REDUCTION	IN	OUT	IN	OUT	IN	OUT
None	5.3	2.4	8.8	10.4	12.1	9.8
1.6	11.4	8.8	14.8	10.4	15.6	8.0
2.0	7.0	2.4	7.8	10.4	12.6	9.6

Table 19 11-NN error rates (%) for 2x2 processor switching functions.

	2X2 CPU		3X3 CPU		4X4 CPU	
FEATURE REDUCTION	IN	OUT	IN	OUT	IN	OUT
None	8.5	2.4	9.3	10.4	15.6	9.6
1.6	20.0	2.88	24.4	24.0	21.1	26.4
2.0	8.0	2.4	8.2	10.4	12.2	9.6

Table 20 k-NN error rates (%) for 4x4 processor switching functions with no feature reduction and case subsampling on Training Set 1.

	K=1		K=5		K=11	
TRAINING SET	IN	OUT	IN	OUT	IN	OUT
Full	7.8	2.4	5.3	2.4	8.5	2.4
Sub 1	8.1	2.4	8.9	2.4	8.4	2.4
Sub 2	9.2	2.4	9.7	2.4	11.6	2.4
Sub 3	9.6	2.4	10.0	2.4	10.2	2.4

Table 21 LDA error rates (%) for 2x2 processor switching functions.

	2X2 CPU		3X3 CPU		4X4 CPU	
FEATURE REDUCTION	IN	OUT	IN	OUT	IN	OUT
None	12.1	21.6	15.4	26.4	18.0	27.2
1.6	21.7	34.4	25.1	34.4	24.7	40.0
2.0	26.6	32.8	30.5	34.4	28.9	34.4

Table 22 LDA error rates (%) for 2x2 processor switching functions with no feature reduction and case subsampling on Training Set 1.

	TEST SET	
TRAINING	IN	OUT
Full	12.1	21.6
Sub 1	11.9	21.6
Sub 2	12.8	21.6
Sub 3	12.1	21.6

Table 23 Benchmarking Cost Summary, 2x2 processors.

BENCHMARK	NUMBER OF BENCHMARKS*	CPU MINUTES**
Training Set 1	2197	1560
Sub 1	1521	1081***
Sub 2	1183	841***
Sub 3	845	600***
Test Set 1	729	344
Test Set 2	500	224

* Recall that each benchmark is four measurements.
** A CPU-minute is the wallclock time in minutes multiplied by the number of processors used.
*** These times are estimated by multiplying the percent reduction by the time for the original training set.

Table 24 Prediction penalty analysis for the data mining methods trained with the complete Training Set 1 measured on the Inside Test Set.

MEASURE	PLAPACK	LDA	5-NN	TREE, S=2.0
Error rate	27.3%	12.10%	5.35%	5.9%
Av. Penalty	17.20%	10.12%	6.93%	5.59%
Stdev(penalty)	.096	.083	.065	.048
Max(penalty)	41.42%	36.23%	26.09%	24.07%
Min(penalty)	0.02%	0.01%	0.02%	0.08%

Table 25 Prediction penalty analysis for the decision tree methods trained with the training set Sub 2 measured on the Inside Test Set.

MEASURE	PLAPACK	TREE, S=1.6
Error rate	27.3%	5.6%
Av. Penalty	17.20%	8.9%
Stdev(penalty)	.096	.083
Max(penalty)	41.42%	34.13%
Min(penalty)	0.02%	0.08%